PRACTICAL GUIDE TO SELINUX

PRODUCED BY: CULLEN REZENDES

CXR5971@RIT.EDU.COM

IF YOU COME ACROSS ANY PROBLEMS, SEE SECTION 7 FOR POSSIBLE SOLUTIONS OR CONTACT ME AT CXR5971@RIT.EDU

Table of Contents

About This File

This file was created for the benefit of all teachers, students, security engineers, system administrators, and anyone else wanting to use SELinux for securing a Linux system.
The entirety of the contents within this file, and folder, are free for public use.

SECTION 1

Brief Introduction to SELinux

SELinux, which stands for Security-Enhanced Linux, is a Linux Security Module (LSM) that implements Mandatory Access Control (MAC) checks on Linux systems. The project was originally developed by the National Security Agency (NSA) and eventually was integrated into the Linux kernel through the use of Linux Security Modules.

1.1 What is SELinux and Why Should I Care?

SELinux was initially developed to fight the seemingly never-ending problem of relying on developers for application security. Applications can interface with the system at a high-level with potentially risky behavior if misused. For example, improper PHP function calls on a web application could allow a user to execute arbitrary code.

1.2 History of SELinux

Security-Enhanced Linux (SELinux) was developed by the National Security Agency (NSA) in an attempt to provide better operating system security that could account for the significant lack of security consideration from application developers. Most modern operating systems utilize some form of access control to prevent unauthorized actions to occur on a system. Access control mechanisms generally take the form of allowing/preventing subjects from accessing objects based on some preconceived notion of what should be allowed. The most common form of built-in access control from operating systems is known as Discretionary Access Control (DAC). DAC simply allows owners of objects (files, programs, services, etc.) specify who may access those objects. This approach seems reasonable enough, but starts to fall apart as computing is utilized in more distributed and high usage environments. For example, a web hosting company

that has hundreds of developers and servers may not want their developers to (accidentally or purposefully) give access to a critical resource they are working on. A developer may own some critical company piece of data that should be more carefully managed.
Mandatory Access Control (MAC) was formally defined in the Trusted Computer System Evaluation Criteria in 1989 by the National Security Computer Center as a way to let the system manage access control. The development of a MAC implementation began with NSA and was eventually named SELinux. SELinux would finally be implemented into the Linux kernel with the introduction of Linux Security Modules, which allowed for a security module, SELinux, to monitor and perform actions when the kernel attempts to perform some action.

1.3 SELinux Security

SELinux combines many different security concepts to implement a secure access control system. There are a number of concepts that enable the system to make decisions based on written policies. We will discuss type enforcement, role-based access control, and (briefly) multi-level security. Overall, type enforcement and role-based access control form the major components for the Mandatory Access Control implementation.

1.3.1 Type Enforcement (TE)

Type enforcement (TE) is the main driving force behind SELinux's implementation of Mandatory Access Control. Every artifact on an SELinux enabled system is labeled with a specific type. The general idea is that there are subjects and objects with some level of access between them. SELinux policy consists of written rules to manage the access between subjects and objects. More hands-on exposure to this concept will be shown in the SELinux policy section.

1.3.2 Role-Based Access Control (RBAC)

Role-Based Access Control is another component in SELinux's type enforcement architecture. Roles allow an administrator to define identifiers that are granted access to transition into a specific domain. We haven't really discussed domains yet, but SELinux "role" is essentially analogous to roles in a company hierarchy. A financial administrator should only be able to access financial documents, just as the webserver role in SELinux should really only be able to access web related files and related processes (PHP, Database, etc.).

1.3.3 Multi-Level Security (MLS)

SELinux also offers a form of Multi-Level Security which is primarily used in government applications to enforce the concept of classification levels. There are a number of important attributes regarding MLS that is implemented such as the Bell-La Padula model, which states that a process can read objects below its security level and write objects that are above its security level. However, the process cannot write objects below its security level and read objects above its security level. MLS is beyond the scope of this document, as its complexity is high and usage is generally seen in government spaces. The notion of security levels will be seen in many parts of

SELinux, as listing the SELinux information about any object will show its security level. Unless we turn on the MLS features of SELinux, MLS will not play a role in our interactions.

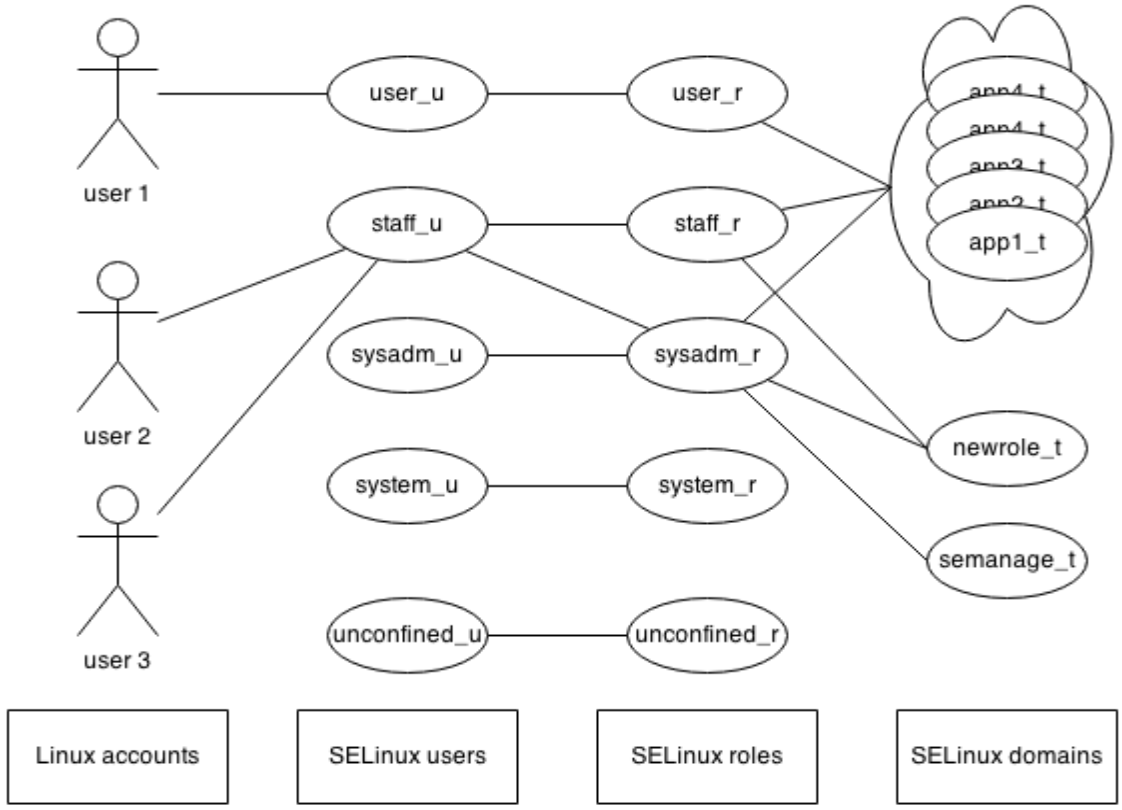
SECTION 2

SELinux Building Blocks

SELinux has a number of contexts that are important to understand before we can truly understand it's policy and enforcement infrastructure. These contexts are denoted as user, role, type, and level (MLS). Additionally, SELinux types are usually associated with some sort of domain which will also be discussed.

2.1 Users

The SELinux user context is not different from a Linux user when it comes to access control. An SELinux user is an identity used by policies to authorize for specific roles and security levels (if enabled). It's important to note that SELinux users are not Linux users, and an administrator needs to defined SELinux users that map to Linux users. So why can't we just use Linux users? The reason is for flexibility and immutable enforcement in the SELinux architecture. By allowing SELinux user identities (defined in policies) to be mapped to roles, allows us to easily enforce that the Linux user cannot change their role. Additionally, this allows for multiple Linux users, to be mapped to the same SELinux user which is then granted any number of roles. So, the relationship between Linux users to SELinux users is many to one.



2.2 Roles

Roles are a component of the Role-Based Access Control feature in SELinux. This allows us to assign role identifiers to access certain "domains" in SELinux. We will discuss domains in the section following this one, but essentially it allows us to be more granular in giving access to users on the system. So, not only are there type enforcement checks, (allow rules that grant access from some source type to some destination type), but also role checks to see if the associated user is even allowed to attempt the transition into the destination domain.

2.3 Types

Types are the bread and butter of SELinux type enforcement. This context is a label assigned to an object which can be used in policies to specify what type of access is allowed. SELinux is only concerned with two things in type enforcement, the type (or label) and the object's class (e.g. file, socket, etc.). This allows us to get granular in how we want to define an object's permissions. For example, a user upload directory for a website will be labeled with a type allowing for writing and potentially reading. However, more sensitive directories may not allow for reading or writing, such as configuration files needed for the web server to continue running.

2.4 Domains

Finally, we approach the domains section. Domains are simply types that are assigned to processes. This concept of a domain, allows policies to enforce what actions a process may take purely based on its label. In the example of an Apache server, we have the httpd_t assigned to the httpd binary. Subsequent files associated with the web server (/var/www/html/* files), will

be labeled with similar named labels such as httpd_user_rw_content_t which allows for httpd to read and write to the labeled content.

Domains are usually inherited from the user executing a process, but can also have dedicated domains specified by the system. For example, the passwd utility has its own domain as it needs to be further locked down due to its access of the passwd file. These processes that have associated domains are denoted as confined processes. However, there are also unconfined processes that are not locked down as much as confined processes. SELinux policy rules are still applied to the system, but the lack of rules targeting unconfined processes leads to Discretionary Access Control mechanisms being used almost exclusively.

SECTION 3

SELinux Architecture

Understanding how the SELinux Architecture is setup is crucial to understanding how policies are created and loaded into the system.

3.1 Linux Security Modules

As mentioned in the history section, Linux Security Modules were designed to give configurable modules access to allow/deny actions taken on important kernel objects. If one process attempts to access a file or process it shouldn't, the LSM framework will catch this interaction is terminate it. This inclusion to the Linux kernel provided a generic interface for developers to add their own customized security modules. SELinux is just one of the possible modules that takes advantage of LSMs. Other modules include AppArmor, Smack, TOMOYO, Landlock, and more. We will walk through the following diagram to explain how Linux Security Modules interact with the system.

First, some user space level process makes a request to access a kernel resource. This results in a Linux system call, which transfers control to the kernel space in order to access some resource. There are a number of system calls that would open another Pandora's box of concepts, but some examples include executing programs, making network connections, file I/O, etc. After error checking of the system call at hand, we pass through normal Discretionary Access Controls. This is important as LSM Modules (SELinux) will never be called for if DAC rules catch an invalid access request first. What does this mean in practice? Say that you use the chmod command to change a file's write access to only root. If someone besides root tries to write to the file, SELinux will never be called as the DAC attributes on the target files already block write access.

If DAC does not catch any problems, then we finally reach the LSM hooks phase. This is where multiple generic security hooks have been placed for modules to interact with.
Once again, this resource could be one of several types. LSMs use several types of hooks, which define some access to monitor a type of resource. Each hook is called based on the type of action and contains subsequent information regarding the action. The categories of LSM hooks are listed below:

- Task Hooks
- Program Loading Hooks -
- IPC Hooks
- File System Hooks
- Network Hooks
- Other Hooks

Network Hooks would be useful when socket related functions are being called whereas file system hooks wold be useful when the user process is attempting to access a specific file or inode. It may be of interest to you to do some research into each of the hooks but that is beyond the scope of this guide. At this point, we use the information provided by the LSM hooks in our policy engine. We will talk more about this in the following section, but the LSM hook information is check against a set of rules and the engine returns an answer of yes or no if the action can proceed. If yes, the process continues to the final destination resource/action (e.g. loading a program, accessing a file, etc.).

We now have an understanding of the Linux Security Module architecture that enables SELinux to exist. In the following section, we will drill-down further and see how SELinux exists in of itself.

3.2 SELinux LSM Architecture

Following from the previous section, we will now take a look into the SELinux module architecture to understand how the SELinux module processes access requests.
The LSM information passed to the SELinux module is processed and checked against the loaded SELinux policy. The two main components that are relevant to using SELinux in most practical scenarios is the security server and the access vector cache (AVC). The security server is designed to hold all of the loaded SELinux policies and makes the access decisions depending on the request made from the user.

Once this request has been processed, either a "yes" or "no" will be returned from the security server. This output is stored in the access vector cache (AVC). The AVC is setup to cache SELinux

decisions for quick response in the future. If the SELinux server makes a particular decision on a user accessing a file, it should store that decision so any subsequent attempts to access the file can be quickly remembered and denied. The AVC plays a very important role in troubleshooting and has been a significant factor in quickly generating supplemental policies to quickly allow a specific action. The logs for AVC is stored in /var/log/audit/audit.log. There are a number of utilities designed to One great example of this in practice is allowing a program to ping. Interestingly enough, the apache domain does not have access to ping. When the ping binary is executed, there are a number of permissions it requires such as

4

SECTION

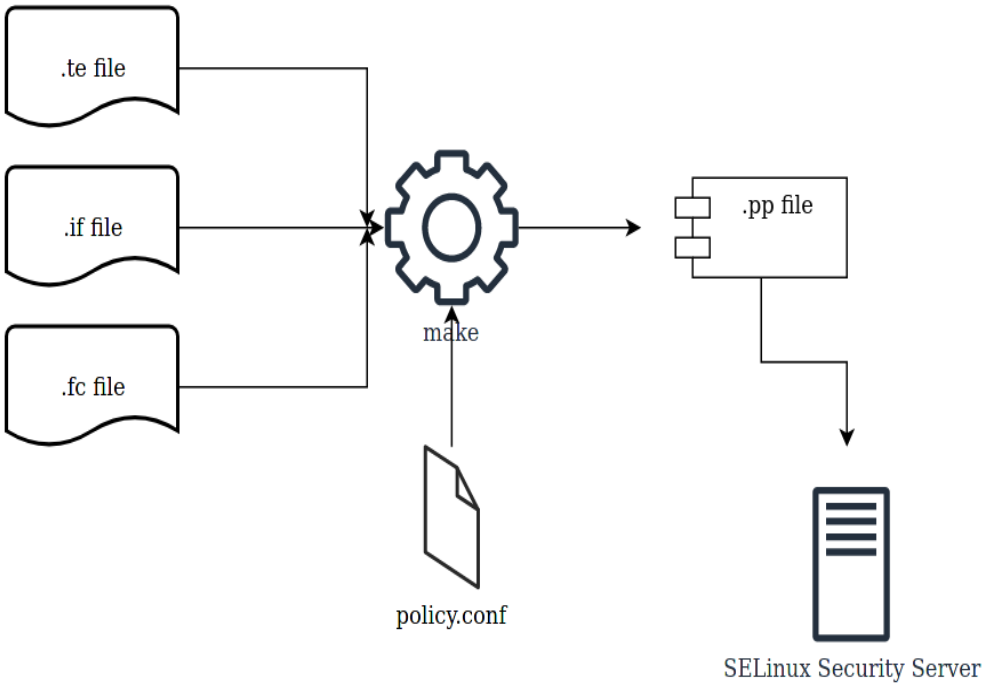
SELinux Policy Architecture

The SELinux Policy Architecture have gone through a number of iterations and forms over the years. The most important and widely used standard for SELinux policy in 2021 (and beyond) is the reference policy. The reference policy is packaged with all the source policy, building, configuration, and documentation files that one could need when deploying SELinux policies. However, SELinux policy becomes a nebulous cloud of confusion in regards to describing what a policy is made of and how to manage it. So, we will break down what SELinux policy "means".

First, there are a number of ways to describe a policy. SELinux policy is generally boiled down to two classifications: monolithic policy or base/loadable modules. The monolithic policy architecture loads all of SELinux policy from a policy.conf file into a singular binary policy file. This has less overhead than the loadable modules architecture and is therefore recommended for embedded systems. In most general cases, the need for flexible policy management leads us away from the monolithic architecture.

Secondly, we have the base/loadable modules policy architecture. This allows SELinux administrators to easily customize which policies are loaded to fit their needs. The standard for SELinux policies is called the reference policy. The core components of the system (e.g. kernel level processes) are loaded into the base module which is always active on the system. The administrator can now create and load separate policies for any context on the system! For example, you could create a policy that is created for a custom application or modify the policy of an existing application such as apache, mariadb, or even raid (mdadm) tools (and more). Many of the policies for popular applications ends up being contributed by system adminstrators and SELinux users

that want to confine the applications and share them. Most operating system developers build their standard SELinux policy set from the reference policy and pre-package it with the distribution download. We will talk more about the policy languages and syntax used in a moment and instead focus on the file format and layout of the policy files from a practical point of view.



One of the most important policy concepts to remember is the deny-all approach of SELinux. If there is not an allow rule explicity allowing some action to occur, then it will be denied. This is useful as we only want to give the bare minimum access necessary for an application to perform its job. However, this has created one of the most frustrating aspects of SELinux policy enforcement. How can someone change a policy module that has already been loaded into the system without breaking everything? We will talk about this is Section [5] but for now we can move on.

4.1 Reference Policy File Structure

The reference policy added a number of policy features that both simplified and enhanced SELinux policy development. More specifically, the reference policy source files exist in three files that can be compiled into a .pp (policy package) file and loaded into the system. We will now take a look at each of the files more in depth.

4.1.1 Type Enforcement (.te)

There is the type enforcement (te) file that contains the main SELinux policy allow rules and definitions. This contains everything needed to allow the SELinux security server the ability to make access decisions. Additionally, various attributes such as types, roles, users, etc. can be defined here. There are a number of rule types and syntax pieces that we will explore more later, but as this guide is not intending to be a comprehensive language guide we will not go in-depth. More understanding of the language syntax and available rules can be found at [INSERT SELINUX NOTEBOOK HERE].

4.1.2 Interfaces (.if)

The interfaces (if) file contains easy to use functions for policies to interact with each other. The best way to describe interfaces is thinking about a function in any programming language. A function is designed to take some input, perform logical operations, and finally return some output. An interface does exactly the same thing! In general, it allows for the administrator to give different levels of access to other policies that may request it.

For example, in the apache policy there is an interface called "apache_admin", which takes a domain and role that the calling policy wants to give access to. This interface will perform all of the appropriate operations to give that domain and role the access needed to properly administer apache (access httpd_t files, start/top the httpd service, search logs, etc.). This intends to make policy development easier by abstracting actions that an administrator wants to perform without needing to write all the rules every time. However, this can also make it difficult to understand exactly how a policy is allowing certain actions without taking an in-depth look at each interface it calls. This issue is alleviated with more in-depth policy analysis tools (apol, sesearch, etc.). We will look at this more in Section [5] when we discuss some general use cases for SELinux.

4.1.3 File Contexts (.fc)

As you may be wondering at this point, how does an SELinux policy know what the difference between bin_t and httpd_t is? There must be some place that defines security contexts on objects. Enter the file contexts (.fc) file. Each policy contains a .fc file that contains all of the context assignments for files and directories. This uses a regular-expression (regex) string to identify what objects (e.g. /etc/apache/*) to target and a corresponding gen_context command that generates the security context (contains user, role, type, security level). If you are writing a policy for a custom application, you will need to add the application's directories and corresponding files to this listing.

4.2 Kernel Policy Language

The kernel policy language comprises the building blocks of SELinux type enforcement policy. There are a number of syntactical concepts to be aware of, and it will be briefly discussed here along with the most important SELinux policy statements to get you started. Note that this guide doesn't aim to be a complete policy language documentation. Instead, we aim to discuss the most common and useful policy statements to get a novice started in the world of policy creation.

SELinux policy language is generally compromised of policy statements that adhere to the kernel policy language standard, along with reference policy m4 macros that aid in building policies (discussed later in this section). In order to target the most important and widely-used kernel policy statements for this guide, we can look at how many of each rule type there is on our system. As you can see from my Centos 8 system, there are significant differences in the rules that

are often used vs. those that appear infrequently. Coupled with some domain knowledge of SELinux, the important rules that we will discuss are the following:

- Types
- Users
- Booleans
- Allow
- Auditallow
- Constraints
- Roles
- Dontaudit

The best way to understand the policy syntax is to dive directly into an example. So, we will look at the apache policy files (.te, .fc. if).

4.2.1 M4 Macros

Before we look directly into the policy, let's discuss what M4 macros are. The introduction of the reference policy architecture not only introduced separate files (.te, .fc, .if) to more easily define a policy module, but it also added multiple macros that act as shortcuts to kernel policy statements. These macros are put in place to ease policy development. Some common macros that will be seen include the following:

- policy_module - This macro is used to name the policy module and give a version to the module for tracking purposes
- gen_tunable - This macros is used to set a boolean's (discussed in the next section) default value

There are many types of m4 macros that perform a variety of tasks on an SELinux-enabled system. They exist in the following directory POLICY_DIR/policy/support.

- loadable_module.spt - Loadable module support
- misc_macros.spt - Generate users, bools and security contexts
- mls_mcs_macros.spt - MLS / MCS support
- file_patterns.spt - Sets up allow rules via parameters for files and directories
- ipc_patterns.spt - Sets up allow rules via parameters for Unix domain sockets
- misc_patterns.spt - Domain and process transitions
- obj_perm_sets.spt - Object classes and permissions

For now, we can move onto the kernel policy statements.

4.2.2 Type

First we address the type kernel policy statement. Simply, this statement creates a new type. The general convention is that there is a binary_t type, binary_exec_t, and any number of types associated with the binary in question. When a user wants to execute a program, they must run a binary with the binary_exec_t type. The reason we have an exec type and a standard type is to ensure secure domain transitions. The problem with running programs is that they often have access to other system resources that our user may not have access too. So, we define the ability for a user to execute the passwd_exec_t, permission for the passwd_exec_t type to access the passwd_t domain, and then we can create allow rules that allow passwd_t to access certain system resources. We will see more about how this works with the type_transition rule but for now we can look at some type examples.
As you can see, we have defined the httpd_t and httpd_exec_t types to be used in our policy. In summary:

- Use the type statement to create a new type label
- Create a new type when you want to define access for some labeled object

4.2.3 User

SELinux user statements are designed to identify a Linux user and subsequently link to an SELinux role (described next). It is important to note that an SELinux user and Linux user are not the same thing. Similar to types, the SELinux user statement defines a label for a Linux user and can link to one or more SELinux roles (which can define type transitions).
Additionally, I will show the login mapping and how a user is mapped to a role. The following two photos shows a login mapping and the subsequent roles assigned to a particular role. We will see in the examples section how we can defined a user-role connection and what impact that has on the system.

```
[root@localhost ~]# semanage login -l

Login Name           SELinux User         MLS/MCS Range        Service

__default__          unconfined_u         s0-s0:c0.c1023       *
demoadmin            demoadmin_u          s0                   *
root                 unconfined_u         s0-s0:c0.c1023       *
testadmin            testadmin_u          s0                   *
```

Login Listing

```
[root@localhost contexts]# semanage login -l

Login Name           SELinux User         MLS/MCS Range        Service

__default__          unconfined_u         s0-s0:c0.c1023       *
root                 unconfined_u         s0-s0:c0.c1023       *
testadmin            testadmin_u          s0                   *
[root@localhost contexts]# semanage user -a -R demoadmin_r demoadmin_u
[root@localhost contexts]# seinfo -u demoadmin_u

Users: 1
   demoadmin_u
[root@localhost contexts]# seinfo -u demoadmin_u -x

Users: 1
   user demoadmin_u roles demoadmin_r level s0 range s0;
[root@localhost contexts]#
```

Roles assigned to root

4.2.4 Roles

As explained in the user section, an SELinux role is used to assign what permissions a certain user should have. Now, roles play a strange part in how SELinux enforces access requests. As we have learned about type enforcement, every program has particular types that define what it can and can't access. Domain transitions operate to allow users to transition control to a particular locked down domain. This means that if a user runs the passwd program, the passwd domain is still in play on that interaction. Even if the user has an unconfined user and role (SELinux rules are not written to affect unconfined users and roles), the subsequent actions are still confined by the passwd domain because we transitioned.

So, what is the point behind roles at all? Well, it's really more of a way to further confined what a particular role (and user), should be able to do. The idea is that we must first check to see what a particular role should do. If a user is assigned to the user_u SELinux user, which is mapped to user_r role, it does not have access to much of the system functionality. For example, the very generic user_r role does not have access to transition to the xdm_t domain (graphical manager domain) then it doesn't matter if there is an allow type rule for the transition. The interaction WILL NOT OCCUR. This exists to allow an SELinux administrator to easily and definitively allow only certain access based on who you are, no matter what subsequent type rules say. An administrator can quickly create a user and assign a particular role that they know won't allow sensitive interactions, even if some application policy says it is okay.

Here is a list of the current roles on the system, as well as a role statement from the apache policy

files. As we can see, there are a number of roles already defined on the system. Additionally, we can see how to define a new role and assign types to it. There is a lot more to defining a new role for the system (it can be somewhat complex) and we will discuss this process more in the example section.

```
[root@localhost cullen]# seinfo -r

Roles: 15
   auditadm_r
   dbadm_r
   guest_r
   logadm_r
   nx_server_r
   object_r
   secadm_r
   staff_r
   sysadm_r
   system_r
   testadmin_r
   unconfined_r
   user_r
   webadm_r
   xguest_r
[root@localhost cullen]#
```

Roles Listed

```
role system_r types httpd_t;
```

Role Defined in Apache Policy

4.2.5 Boolean

Boolean statements were initially created to once again make it easier for SELinux administrators to define access between applications, roles, etc. In practice, these are implemented with the tunable_policy macro. This macro will allow a permission based on the value of a particular boolean. The default value of the boolean is set with gen_tunable at the beginning of a policy file and can be changed on the commandline with the following command:

```
$ setsebool
```

Booleans can help quickly change common policy settings like allowing apache to access certain directories, make outbound network connections, etc. We can get a listing of some of the booleans as such.

```
[root@localhost ~]# semanage boolean --list | grep httpd
awstats_purge_apache_log_files (off  ,  off)  Determine w
httpd_anon_write                (off  ,  off)  Allow Apach
ansfer services. Directories/Files must be labeled public
httpd_builtin_scripting         (on   ,   on)  Allow httpd
httpd_can_check_spam            (off  ,  off)  Allow http
httpd_can_connect_ftp           (off  ,  off)  Allow httpd
rt and ephemeral ports
httpd_can_connect_ldap          (off  ,  off)  Allow httpd
httpd_can_connect_mythtv        (off  ,  off)  Allow http
httpd_can_connect_zabbix        (off  ,  off)  Allow http
httpd_can_network_connect       (on   ,   on)  Allow HTTPD
sing TCP.
httpd_can_network_connect_cobbler (off  ,  off)  Allow HT
er the network.
httpd_can_network_connect_db    (on   ,   on)  Allow HTTPD
r the network.
```

Booleans Listed

4.2.6 Allow

Finally, we reach the allow statement. This is the bread and butter of the type enforcement system and defines what kind of access subjects should have on objects. The format of these rules can look scary, but we will break them down more generally.

So, allow rules exist under a subset of rule types called "Access Vector Rules". An allow rule targets a source type, destination type, some object class, and some perm set. I have not discussed these before as I felt it may serve to confuse users if they didn't have the appropriate SELinux context first. Here is an example of an allow rule, and we will break it down now.

```
allow httpd_t httpd_script_exec_type:file read_file_perms;
allow httpd_t httpd_script_exec_type:lnk_file read_lnk_file_perms;
allow httpd_t httpd_script_type:process { signal sigkill sigstop signul
allow httpd_t httpd_script_exec_type:dir list_dir_perms;

allow httpd_script_type self:process { setsched signal_perms };
allow httpd_script_type self:unix_stream_socket create_stream_socket_pe
allow httpd_script_type self:unix_dgram_socket create_socket_perms;
allow httpd_script_type httpd_t:unix_stream_socket rw_stream_socket_per

allow httpd_script_type httpd_t:fd use;
allow httpd_script_type httpd_t:process sigchld;
```

Some Allow Rules in Apache Policy

First, the source and destination type are easy to understand. We have some initial process or file type that wants to access another process or file type. The self keyword may be used to initially give a particular process access to some functionality that the application may need, such as making network connections, basic file reading access to any files with its own type, creating log messages, etc.

Next, there is an object class that is used to describe the type of object we are attempting access on. This helps to define what action should be taken in the subsequent permission section. There are a number of object classes that have been added with the implementation of the reference policy but I will give numerous examples and a table displaying common object classes with common permissions that are shared. This is not an exhaustive list, and I will redirect any further curiosity to the SELinux Notebook hosted by the SELinux Project Github at (https://github.com/SELinuxProject/selinux-notebook/blob/main/src/object_classes_permissions.md#)

4.2.7 Auditallow

Auditallow functions exactly the same as allow, except now we also record an event in the audit logs. Normally, only denials of access are recorded to keep the alerts focused on only potential malicious behavior. There are a number of reasons to use the auditallow statement but it is mostly seen in troubleshooting a policy.

4.2.8 Constrains

Constrain statements are one of the more useful statements in SELinux policy creation and accounts for a significant problem. SELinux operates in a "deny all, allow some" approach. This is great as everything (barring dontaudit rules) will be logged to /var/log/audit/audit.log. If a policy is blocking an action that needs to be allowed, it is easy to troubleshoot and create a policy

off of the subsequent message. However, in the case that a policy is already allowing an action, one that you don't want to be allowed, how can you deny it? Your only options are to find the allow rule that enablind the action in question, or you can use a constrain! Constrains are the opposite of allow rules. Constrains will limit access based on some conditions you provide it. Generally, these are source type, target type, roles, and/or users.
It's interesting that there were no uses of constrains found on the default Centos 8 SELinux policy. Constrain statements are a bit complex, as they operate somewhat like if-statements so we will explore a hypothetical example here, and then approach a real use in the next section. The constrain statement looks like the following:

4.2.9 Attributes and Type Attributes

I have lumped attribute and type_attribute rules together as they generally work hand in hand. Attributes are labels used to group type labels. The point is to be able to write policy rules to affect a number of different, but closely related types. For example, the httpd policy has a number of different httpd_t content types (httpd_sys_rw_content_t, httpd_sys_content_t, httpd_sys_ra_content_t, and more, but they are all generically content types). We can then use the typeattribute statement to assign a type to the created attribute type.

4.2.10 Dontaudit

The dontaudit rule stops auditing of a denial message that would normally occur. You will see many of these rules in place to stop auditing various activities as it can get annoying. For example, attempting a ping to some IP address makes many access requests including setting up a socket, sending data, receiving data, etc. You may end up getting many alerts for that one ping request, and it may be better to just not alert on some of those access requests. Here is an example of a dontaudit rule.
The usage of dontaudit makes sense, but it can be somewhat annoying if you are attempting to debug why SELinux is not denying functionality, you may never see an alert. As I will discuss in the troubleshooting section, many times an entire policy can be built on alert message output so we will want to know how to get around this. The following command will temporarily disable the dontaudit rules so all of the alerts come through.

```
$ semodule –DB
```

4.3 CIL

The need to make writing SELinux policies easier drew developer's attention to the kernel policy language. CIL stands for Common Intermediate Language and is a high-level representation of the kernel policy language we have been discussing. I will not directly represent all of the language statements here, as many of them are very similar with only minor alterations to semantics. I will discuss the goals of CIL and why you may want to use it.

4.3.1 Goals

CIL was created to easy some of the current issues with SELinux policy.

- **Easy Policy Customization** - Provide an easy method for adding additional rules that will remain separate from the main policy modules on the system, thus not breaking updates or dependencies
- **Simple Syntax for Easy Modeling** - SELinux policies have become very complex and new applications are constantly getting policies written for them. It can be very hard to model relationships and truly analyze what is going on in the system. Having a simpler syntax makes modeling easier
- **No Binary Format** - One of the issues with policy modules is the need to compile type enforcement files into a policy package which exists in a binary format. CIL aims to eliminate the need, so source policy files become more common, the kernel can interpret it

4.3.2 Why Use CIL?

The question of why should you use CIL depends on what you need to do. If you are doing a complete setup of SELinux on a new system, I would recommend tuning policies to how you would like from the kernel policy files. However, if you want to make some quick fixes and modifications such as constrains (which are troublesome to get working in kernel policy), then CIL is your best bet. One of the best features of CIL is you can extract any loaded policy module into its CIL version. Once you do that, you can make any modifications to the policy file and reload it. You cannot get the source files from an policy package file as it is already converted to its binary format. Currently, there is a lack of good resources demonstrating CIL usage. There is a guide on the SELinux Notebook in the references section but it lacks clear examples and mainly gives an overview of what might be possible but I have not seen it in practice very often.

5

SECTION

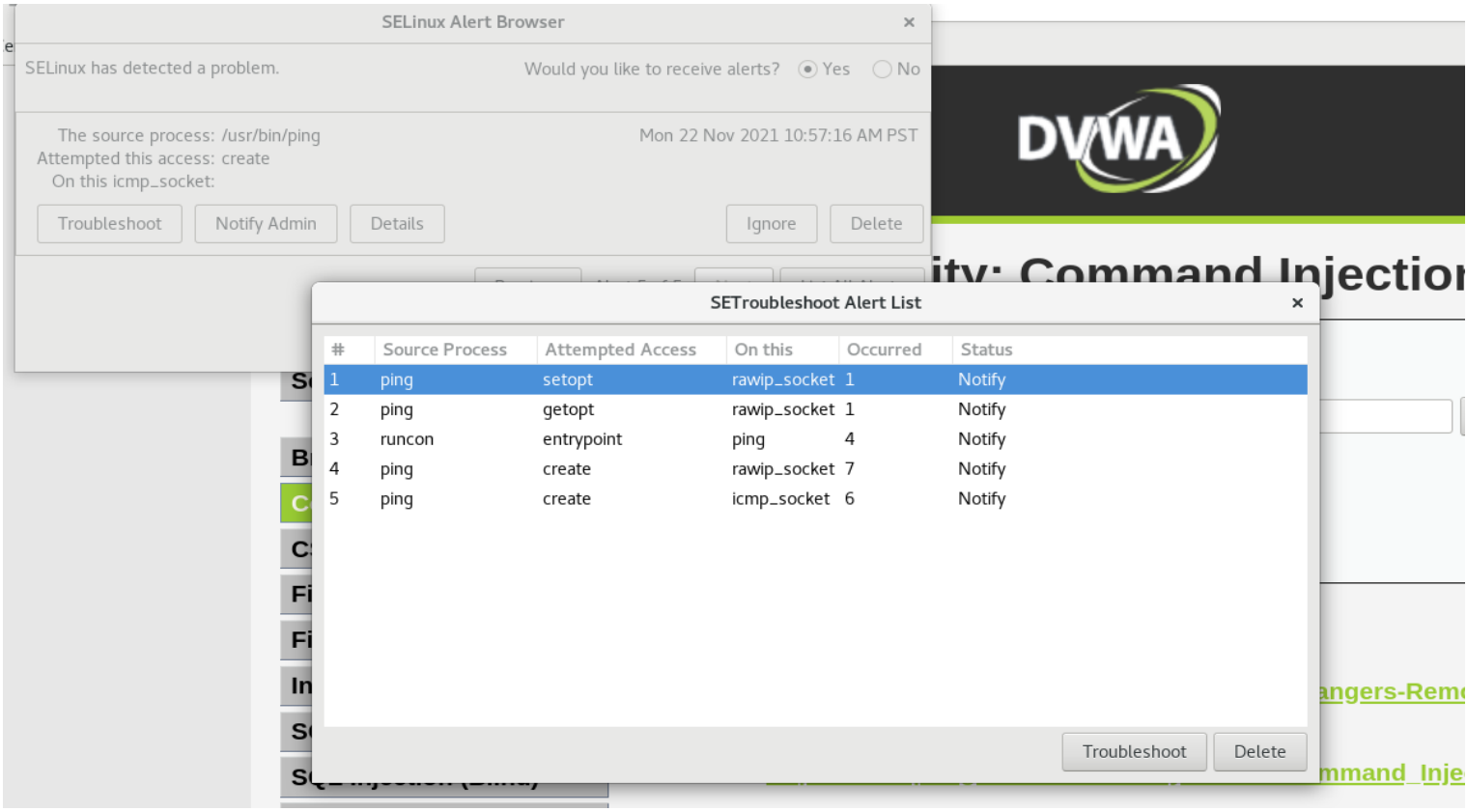
SELinux Administration & Use Cases

At this point, all of the major SELinux concepts have been explored but how to actually administer an SELinux system in real-world scenarios has been avoided. This section sets out to accomplish some of the main configuration use-cases one may have for SELinux.

5.1 Troubleshooting Policy

Troubleshooting SELinux policy may be one of the most important concepts to understand in modern day SELinux usage. The complexities for writing SELinux policy can turn people away from implementing SELinux on their systems. However, the SELinux developers have worked to ensure that correcting denials can be as easy as one or two commands. The audit2allow tool was created to allow an SELinux administrator to transform audit logs into full-blown policies!

As an example, we will be using Damn Vulnerable Web Application as it provides a good basis for insecure coding practices we are trying to mitigate. Here we can see that when we attempt to ping an IP address (a good action we want), and inject a command (a bad action we want to prevent), the ping fails but the command succeeds! We will discuss troubleshooting the pinging first and resolving the command injection comes in the modifying a policy section. So we have a good action that is being denied. We can look at our audit logs and see exactly what is happening here. There are generally two ways to view the audit logs, you can utilize the graphical troubleshooting tool or viewing the logs directly at /var/log/audit/audit.log. We will explore both options.



Troubleshooter GUI - Ping Alerts

```
[root@localhost BUILD]# ausearch -c 'ping'
----
time->Tue Oct 12 10:44:32 2021
type=PROCTITLE msg=audit(1634060672.965:609): proctitle=70696E67002D63003400382E382E382E38
type=SYSCALL msg=audit(1634060672.965:609): arch=c000003e syscall=41 success=no exit=-13 a0=2 a1=2 a2=1
3=7ffda812a0c0 items=0 ppid=38952 pid=38953 auid=4294967295 uid=48 gid=48 euid=48 suid=48 fsuid=48 egid=
8 sgid=48 fsgid=48 tty=(none) ses=4294967295 comm="ping" exe="/usr/bin/ping" subj=system_u:system_r:http
_t:s0 key=(null)
type=AVC msg=audit(1634060672.965:609): avc:  denied  { create } for  pid=38953 comm="ping" scontext=sys
em_u:system_r:httpd_t:s0 tcontext=system_u:system_r:httpd_t:s0 tclass=icmp_socket permissive=1
----
time->Tue Oct 12 10:44:32 2021
type=PROCTITLE msg=audit(1634060672.966:610): proctitle=70696E67002D63003400382E382E382E38
type=SYSCALL msg=audit(1634060672.966:610): arch=c000003e syscall=41 success=yes exit=3 a0=2 a1=3 a2=1 a
=7ffda812a0c0 items=0 ppid=38952 pid=38953 auid=4294967295 uid=48 gid=48 euid=48 suid=48 fsuid=48 egid=4
 sgid=48 fsgid=48 tty=(none) ses=4294967295 comm="ping" exe="/usr/bin/ping" subj=system_u:system_r:httpd
t:s0 key=(null)
type=AVC msg=audit(1634060672.966:610): avc:  denied  { net_raw } for  pid=38953 comm="ping" capability=
3  scontext=system_u:system_r:httpd_t:s0 tcontext=system_u:system_r:httpd_t:s0 tclass=capability permiss
ve=1
type=AVC msg=audit(1634060672.966:610): avc:  denied  { create } for  pid=38953 comm="ping" scontext=sys
em_u:system_r:httpd_t:s0 tcontext=system_u:system_r:httpd_t:s0 tclass=rawip_socket permissive=1
----
```

Ausearch Utility - Ping Alerts

In the graphical SETroubleshoot utility, we see a number of errors related to the ping process. Many of these errors have to do with setting options and creating a socket for ping. For each error we can run commands that the GUI interface tells us will fix the issue. This will create a policy module for loading. I will not use the GUI here because I want to put everything on one policy file. In the command line interface, we can utilize the ausearch utility or simply grep. As we can see, there are a number of denial messages for the various actions that ping was attempting. We can use the audit2allow utility to transform this into a policy.

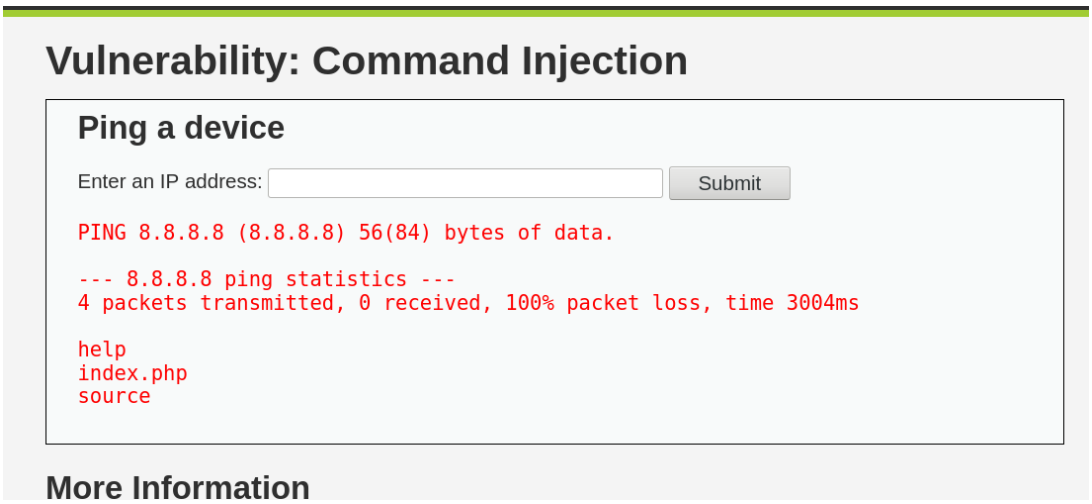
```
[root@localhost BUILD]# ausearch -c 'ping' --raw | audit2allow -a -M apache_ping
******************** IMPORTANT ***********************
To make this policy package active, execute:

semodule -i apache_ping.pp

[root@localhost BUILD]# ls
apache_ping.pp  selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49
apache_ping.te  selinux-policy-contrib-73a88dc7435b803ba860e8938c9611dd62ef6d5c
```

Audit2allow Utility - Creating a New Module

Now that we have created a policy module, we can load it with semodule and clear the audit logs.

```
$ semodule -i apache_ping.pp
$ > /var/log/audit/audit.log
```

When we retry our ping attempt, you may notice that we have a ping attempt! But there is one problem, all of our packets were lost.



Pinging but Packet Loss

If we go back to the audit logs, there are no further denial messages. Once again, the trick here is noted in other resources but there are few examples using it. We must temporarily disable dontaudit rules! As it turns out, there are a number of dontaudit rules in the policy that specify not to audit certain actions, like net_raw read/write capabilities. There is some logic in the idea that we don't want to alert on every network interaction the system has, that would be chaos for the audit system. However, this does make it much more difficult for troubleshooting, so its best to turn this off using the semodule command again.

```
$ semodule –DB
```

When we retry our ping, we get a load of messages for pinging! We can create another policy module based on these and then combine the two (so we aren't loading more modules than necessary). We can re-run the same command as earlier, renaming the policy so we don't overwrite our previous one. We can see there was one small change, namely giving read/write capabilities to the net_raw object. I simply took the changes from the new policy and inserted it in the old type enforcement file. I can now rebuild this module with the SELinux provided compile Makefile located at /usr/share/selinux/devel/Makefile. The resulting module can be installed, which will automatically overwrite the previous instance of this policy. It's also important to note that user loaded modules will be loaded at a higher priority than other modules. This means that rules here will be processed first over anything specified in other modules.

```
module apache_ping 1.0;

require {
        type httpd_t;
        class icmp_socket create;
        class capability net_raw;
        class rawip_socket { create getopt setopt };
}

#============= httpd_t ==============
allow httpd_t self:capability net_raw;
allow httpd_t self:icmp_socket create;
allow httpd_t self:rawip_socket { create getopt setopt };
~
~
~
"apache_ping.te" 14L, 328C
```

First Module Created

```
module apache_ping2 1.0;

require {
        type httpd_t;
        class rawip_socket { read write };
}

#============= httpd_t ==============
allow httpd_t self:rawip_socket { read write };
~
~
```

Second Module Created

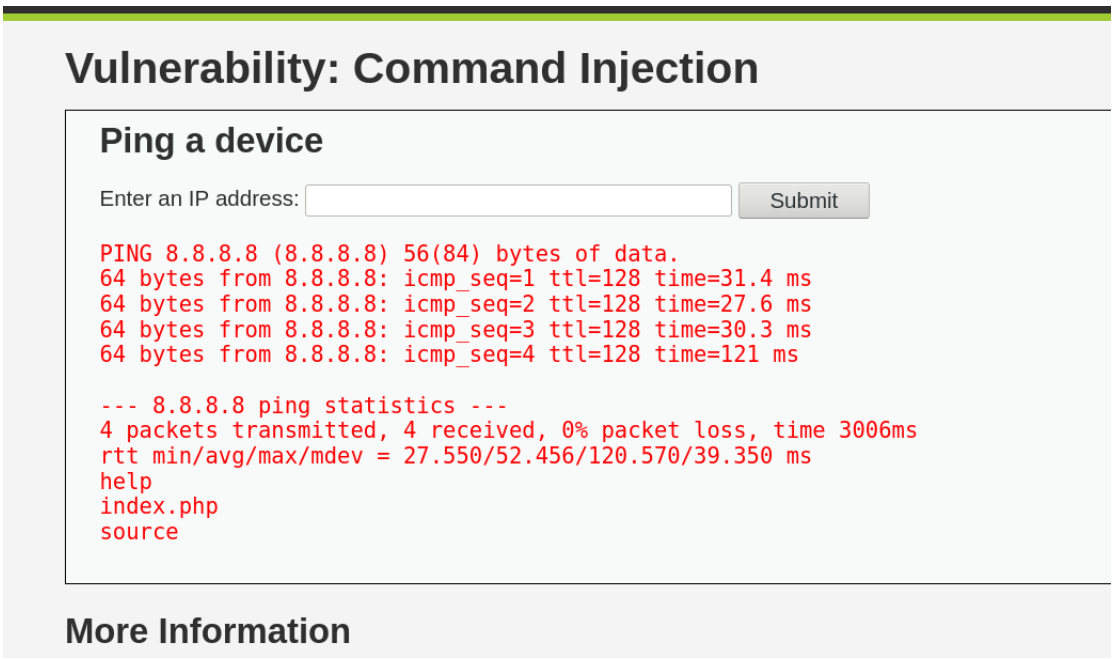
```
module apache_ping 1.0;

require {
        type httpd_t;
        class icmp_socket create;
        class capability net_raw;
        class rawip_socket { read write create getopt setopt };
}

#============= httpd_t ==============
allow httpd_t self:capability net_raw;
allow httpd_t self:icmp_socket create;
allow httpd_t self:rawip_socket { read write create getopt setopt };
~
```

Final Policy

As a reminder, we load the policy module with the semodule command.

```
$ semodule -i apache_ping.pp
```

As you can see, we have ping working! We will get rid of that nasty command injection later.



Final Policy

5.2 Configuring Users and Roles

The Role-Based Access Control (RBAC) component allows an administrator to have fine-grained control over a user's access. As explained in earlier sections, SELinux utilizes a Linux user->SELinux User->SELinux Role architecture that allows a Linux user to get any number of permissions assigned to any number of roles. How do we control what our users are assigned to? How do we create our own users? How do we create our own roles? How can I check what access a specific role has? Fret not, this section will answer all of these questions.

First, let's get an understand of a system in it's default configuration. We will be using a number of our SELinux analysis tools outlined earlier. Using seinfo, we can print the SELinux users, SELinux roles, relationships between SELinux users and roles, and finally what types a role can access.

```
[root@localhost cullen]# seinfo -u

Users: 9
   guest_u
   root
   staff_u
   sysadm_u
   system_u
   testadmin_u
   unconfined_u
   user_u
   xguest_u
```

Listing SELinux Users

```
[root@localhost cullen]# seinfo -r

Roles: 15
   auditadm_r
   dbadm_r
   guest_r
   logadm_r
   nx_server_r
   object_r
   secadm_r
   staff_r
   sysadm_r
   system_r
   testadmin_r
   unconfined_r
   user_r
   webadm_r
   xguest_r
[root@localhost cullen]#
```

Listing SELinux Roles

```
[root@localhost cullen]# seinfo -u user_u -x

Users: 1
   user user_u roles user_r level s0 range s0;
[root@localhost cullen]# seinfo -u sysadm_u -x

Users: 1
   user sysadm_u roles sysadm_r level s0 range s0 - s0:c0.c1023;
[root@localhost cullen]# seinfo -r sysadm_r -x

Roles: 1
   role sysadm_r types { telepathy_mission_control_cache_home_t config_
_t openshift_var_lib_t fsadm_t chrome_sandbox_nacl_t xdm_home_t winbind
 chfn_t mail_home_rw_t cache_home_t hwloc_dhwd_t ssh_t iceauth_home_t g
t telepathy_logger_cache_home_t antivirus_home_t postfix_postdrop_t san
home_bin_t dcc_dbclean_t mpd_user_data_t postfix_postqueue_t sysadm_seu
home_t system_mail_t traceroute_t prelink_t bootloader_t dmidecode_t sa
pathy_mission_control_data_home_t netlabel_mgmt_t usbmodules_t systemd_
cronjob_t vpnc_t user_tmp_t certwatch_t git_user_content_t tvtime_home_
_t load_policy_t openvpn_t oddjob_t updpwd_t mail_home_t chkpwd_t fetch
k_home_t polipo_cache_home_t newrole_t nscd_t amanda_recover_t iptables_
ssion_control_home_t mailman_mail_t irc_home_t sandbox_x_t iotop_t gstr
t kpatch_t vpbind_t vmware_t cdcc_t user_mail_t telepathy_sunshine_home
```

Listing Roles Associated with user_u SELinux User

Generally, an SELinux administrator can confidently stick to the roles provided on the system, applying SELinux user labels (that correspond to SELinux roles), as they see fit. However, there may be case where a custom role should be defined and further confined. We will now create a new role based on the user_r role. This should give a good basis for how to create a role and apply it in practice.

First, we must ensure we have downloaded the source policy files that we want on the system by default. As demonstrated earlier, we will perform the following commands to download the policy:

```
$ dnf download --source selinx-policy
$ rpm -qi selinux-policy
$ rpm -i selinux-policy-X.XXXXX
$ rpmbuild -bp ~/rpmbuild/SPECS/selinux-policy.spec
```

Now that we have ensured we have the SELinux policies for our system, we will navigate to the /root/rpmbuild/BUILD/ directory and note there are two folders. My general understanding is that the contrib folder is mainly add-ons to the SELinux policy (services and other applications that aren't default), whereas the other folder contains many of the core modules. If we navigate to the non contrib directory, then drill down to the roles folder as shown.

```
[root@localhost roles]# pwd
/root/rpmbuild/BUILD/selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/roles
[root@localhost roles]# ls
auditadm.fc  logadm.te     staff.fc   sysadm_secadm.fc   unconfineduser.if
auditadm.if  metadata.xml  staff.if   sysadm_secadm.if   unconfineduser.te
auditadm.te  secadm.fc     staff.te   sysadm_secadm.te   unprivuser.fc
logadm.fc    secadm.if     sysadm.fc  sysadm.te          unprivuser.if
logadm.if    secadm.te     sysadm.if  unconfineduser.fc  unprivuser.te
[root@localhost roles]#
```

Sysadm Role

Let's copy all of the sysadm role files (.te .if .fc) so we can create our own role based on this role. I copied these files and placed them in an entirely different directory to avoid any compilation conflicts. I renamed everything that used to say sysadm to demoadmin.

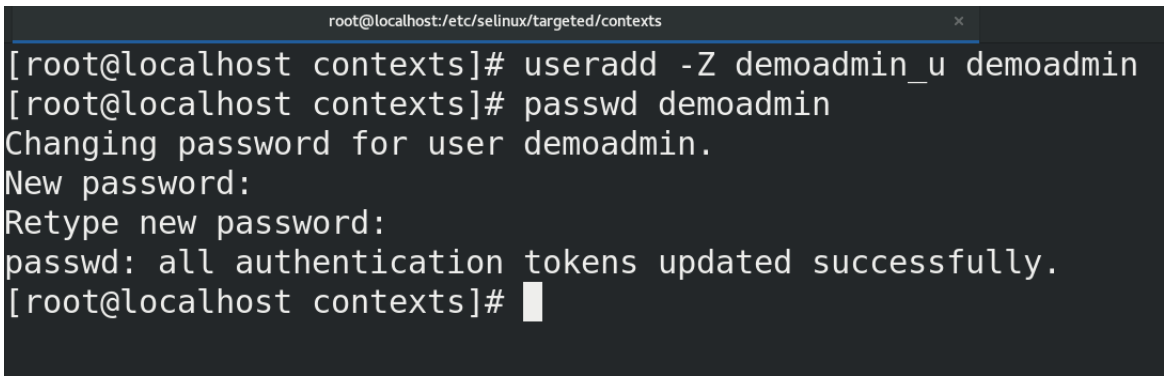
```
[root@localhost new_admin]# ls
demoadmin.fc  demoadmin.if  demoadmin.te
[root@localhost new_admin]# make -f /usr/share/selinux/devel/Makefile
Compiling targeted demoadmin module
Creating targeted demoadmin.pp policy package
rm tmp/demoadmin.mod.fc tmp/demoadmin.mod
[root@localhost new_admin]# semodule -i demoadmin.pp
[root@localhost new_admin]#
```

New Demo Admin Role

We can now create a Linux user that will be assigned to the demoadmin SELinux user and role. After that, we must make a few changes to the SELinux configuration to allow the user to properly login and adopt the correct file labels. First, we edit the default_type file located at /etc/selinux/-targeted/contexts. This file specifies the default type for each SELinux role. We add a record for the demoadmin_r role. Furthermore, we need to transition from the system_r to the demoadmin_r role. The reasoning for this is that when the system boots in and no one is logged in, the system_r role is active. We need to make it clear that the system_r role can transition to the demoadmin_r role and subsequent types. The file in question now is the demoadmin_u user file in the users folder in /etc/selinux/targeted/contexts. The easiest way to do this is to copy one of the existing user files, particular sysadm_u and rename the copy to demoadmin_u. We can actually edit this file to disallow access such as SSH, XDM (graphical login, same as GDM), etc.

```
[root@localhost contexts]# semanage login -l

Login Name           SELinux User         MLS/MCS Range        Service

__default__          unconfined_u         s0-s0:c0.c1023       *
root                 unconfined_u         s0-s0:c0.c1023       *
testadmin            testadmin_u          s0                   *
[root@localhost contexts]# semanage user -a -R demoadmin_r demoadmin_u
[root@localhost contexts]# seinfo -u demoadmin_u

Users: 1
   demoadmin_u
[root@localhost contexts]# seinfo -u demoadmin_u -x

Users: 1
   user demoadmin_u roles demoadmin_r level s0 range s0;
[root@localhost contexts]#
```

Assigning Role to User



Created Linux User

```
auditadm_r:auditadm_t
secadm_r:secadm_t
sysadm_r:sysadm_t
staff_r:staff_t
unconfined_r:unconfined_t
user_r:user_t
testadmin_r:testadmin_t
demoadmin_r:demoadmin_t
~
~
~
```

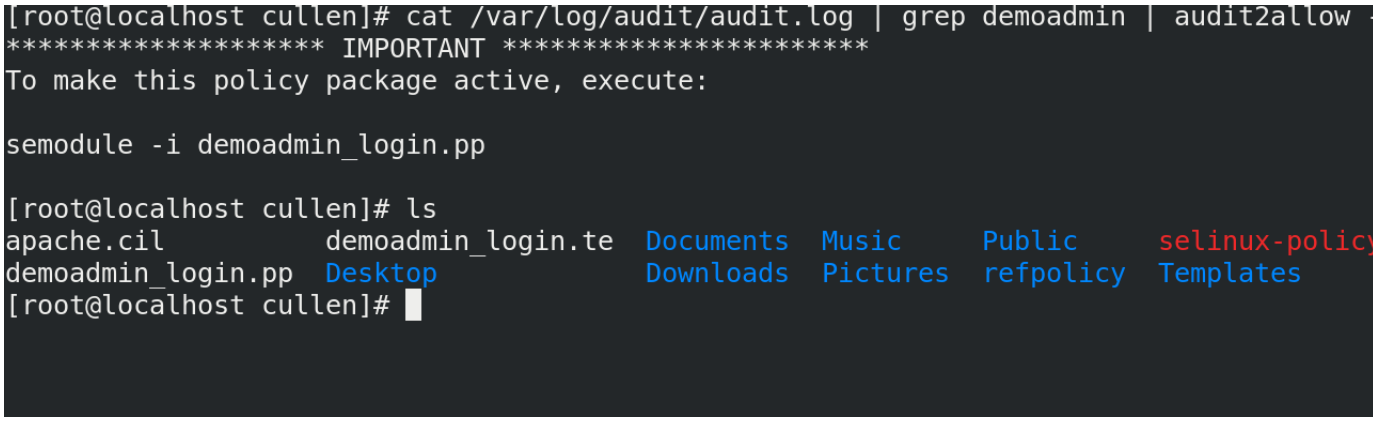
Creating Type Transition - /etc/selinux/targeted/contexts/default_type file

```
system_r:local_login_t:s0       demoadmin_r:demoadmin_t:s0
system_r:remote_login_t:s0      demoadmin_r:demoadmin_t:s0
system_r:sshd_t:s0              demoadmin_r:demoadmin_t:s0
system_r:cockpit_session_t:s0           demoadmin_r:demoadmin_t:s0
system_r:crond_t:s0             demoadmin_r:demoadmin_t:s0
system_r:xdm_t:s0               demoadmin_r:demoadmin_t:s0
system_r:init_t:s0              demoadmin_r:demoadmin_t:s0
demoadmin_r:demoadmin_su_t:s0           demoadmin_r:demoadmin_t:s0
demoadmin_r:demoadmin_sudo_t:s0         demoadmin_r:demoadmin_t:s0
system_r:initrc_su_t:s0         demoadmin_r:demoadmin_t:s0
demoadmin_r:demoadmin_t:s0              demoadmin_r:demoadmin_t:s0
demoadmin_r:demoadmin_su_t:s0           demoadmin_r:demoadmin_t:s0
demoadmin_r:demoadmin_sudo_t:s0 demoadmin_r:demoadmin_t:s0
```

Creating a New Users File for demoadmin -
/etc/selinux/targeted/contexts/users/demoadmin

We can now attempt to login, but there seems to be an issue. Everytime we try to login, we get brought back to the login screen. So, there is something wrong here. Logging back into our root

user and looking at the logs, we find alerts for gdm transitions. Gdm is the graphical user environment, and it seems like we never included an allow rule in our copied sysadm_r policy! The sysadm_r does not have an allow rule to access the graphical desktop, EVEN THOUGH their user file says it is allowed. So, we can use our audit2allow troubleshooting skills to account for this. We could once again combine it all into one policy, but here I choose not to just to demonstrate that it works the same.



Creating a New Policy for GDM

```
module demoadmin_login 1.0;

require {
        type xdm_t;
        type demoadmin_t;
        class process transition;
}

#============= xdm_t ==============

#!!!! This avc can be allowed using the boolean 'xdm_sysadm_login'
allow xdm_t demoadmin_t:process transition;
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"demoadmin_login.te" 13L, 250C
```

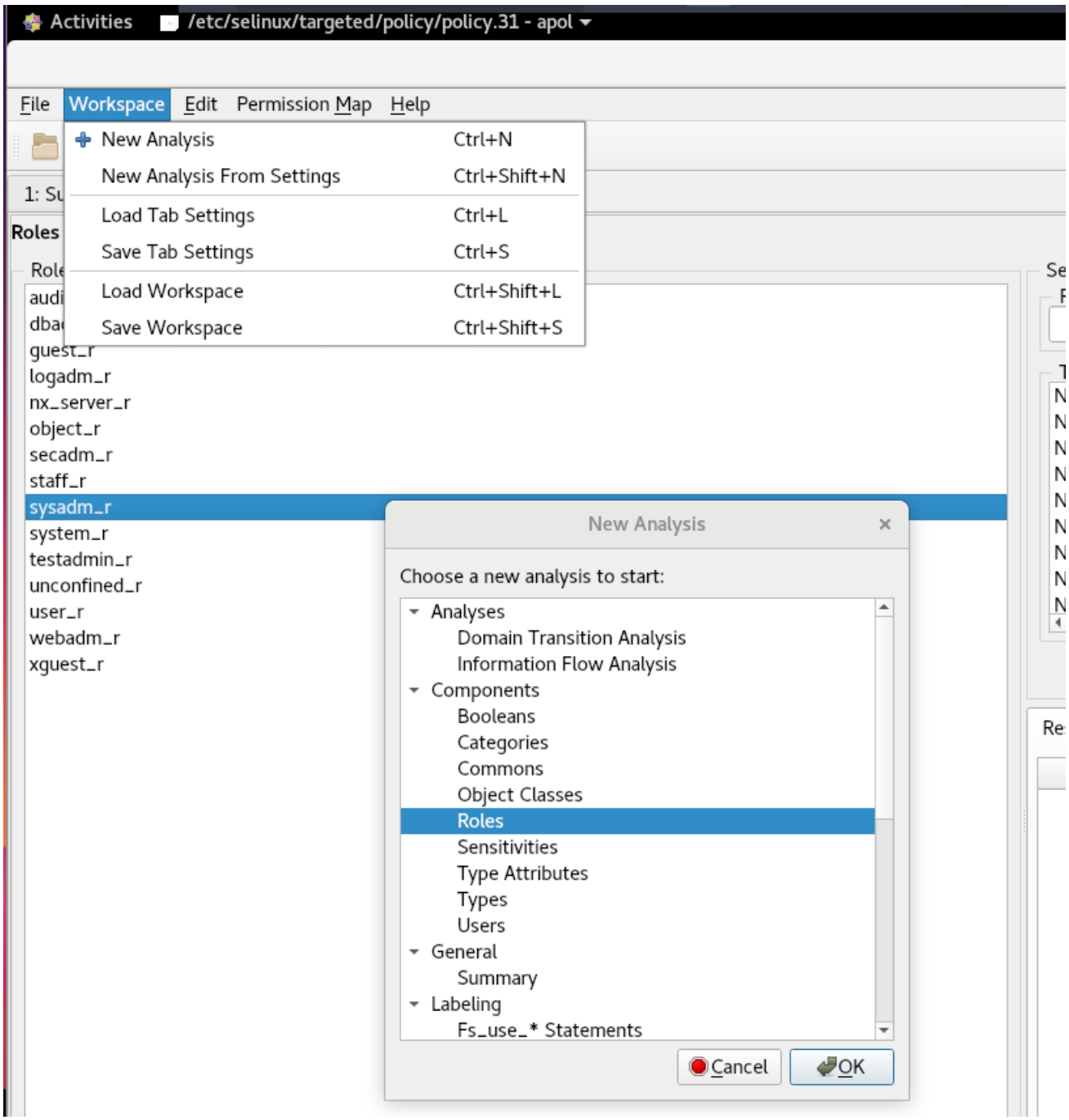
Type Enforcement File Generated

Once we load this policy, we attempt to login again. We are now successful and we can see the file labels associated with our user is on our home directory. Now that we know it is working, we can go back and take out rules or even add constrain rules to lock down our role.

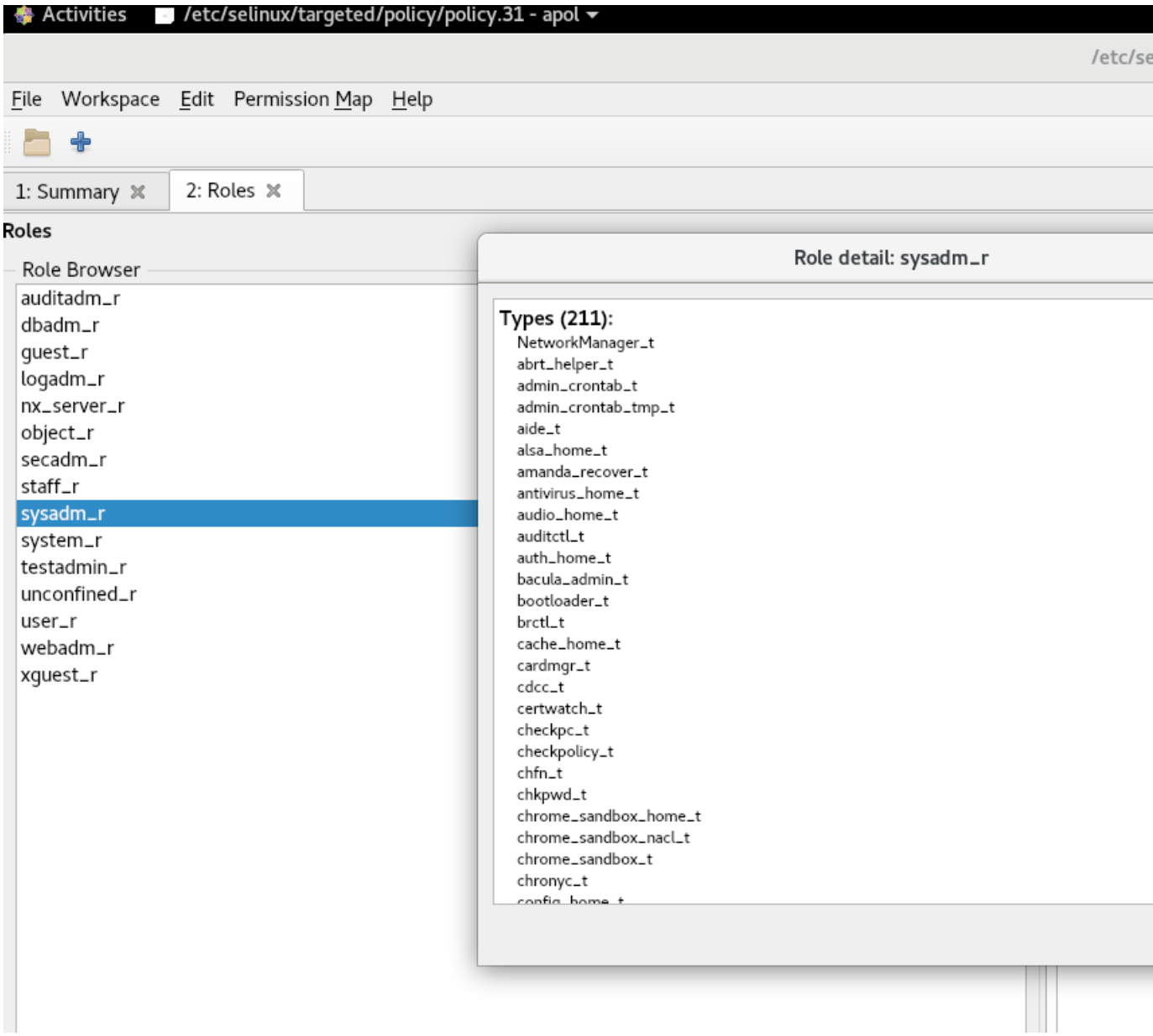
```
demoadmin@localhost:~
File  Edit  View  Search  Terminal  Help
[demoadmin@localhost ~]$ ls -lZ
total 0
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:user_home_t:s0  6 Nov 22 02:20
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:user_home_t:s0  6 Nov 22 02:20
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:user_home_t:s0  6 Nov 22 02:20
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:audio_home_t:s0 6 Nov 22 02:20
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:user_home_t:s0  6 Nov 22 02:20
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:user_home_t:s0  6 Nov 22 02:20
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:user_home_t:s0  6 Nov 22 02:20
drwxr-xr-x. 2 demoadmin demoadmin demoadmin_u:object_r:user_home_t:s0  6 Nov 22 02:20
[demoadmin@localhost ~]$ id -Z
demoadmin_u:demoadmin_r:demoadmin_t:s0
[demoadmin@localhost ~]$
```

5.3 Viewing Current Policy

In many cases, we may want to see what the current policy on our system is. Understanding how policies are setup by default will enable an administrator to understand later errors or gaps in access control. There are many ways to view SELinux policies. The most user friendly tool is apol, which is a graphical policy analysis tool. You can perform many different actions such as searching for specific domain transitions, what types a user is assigned to, what roles exist on the system, etc. Once you select your policy file, located at /etc/selinux/targeted/policy/policy.X, you can perform any number of analysis procedures on it. I demonstrate viewing roles and associated types below.



Looking at Roles Registered on the System



Types Associated with a Role

5.4 Modifying a Currently Installed Policy

One of the biggest problems I have personally seen with SELinux and is rarely addressed in any other resources is how to efficiently modify a currently installed policy. There are a number of issues associated with modifying installed policies. First, we have discussed SELinux's "deny all, allow some" approach. This approach makes adjusting for denials quite simple as we have seen in the Troubleshooting Policy Section. We are able to take an alert, or a number of alerts, and convert it into a functional policy package (.pp) to be loaded. However, what do you do if some action is allowed that you don't want it to be? The answer may seem obvious if you have been reading this document in its entirety but it is not blatantly clear. Additionally, trying to edit policies can result in unexpected consequences, such as other policies that are relying on the policy you are editing to now have some issues if certain definitions are removed. This has been slightly remedied by the introduction of the reference policy, which sets out to divide policy functions into different files (interface files are used to allow for other policies to access

functionality).

There are two methods that I would recommend to deal with this problem. One method is to download the targeted policy designed for the system you are currently using, modify the accompanying policy module files, and reload the policy at a higher priority. The other method is to use a constrain statement, which you can write in a single policy module to get exactly the functionality you want to restrict. I will discuss both of them now!

5.4.1 Reloading Modified Policy Module

Reloading a modified policy module may seem simple when first approaching the problem, but you'd be surprised at how difficult it is to see people discussing this in other SELinux resources. The first problem is retrieving the appropriate SELinux policies for the system that you are currently running. It is critical that you retrieve the SELinux policies that are closest to the system you are running, as every Linux distribution has built and modified the reference policy to meet their specific needs. If you attempt to download the base reference policy and load that into the system, there is a good chance that it will not work. The policies are just too far from what the system is currently running with to be fully compatible. On RPM based systems, the solution is to use the following commands (which we used in multiple previous sections)
dnf download –source selinx-policy rpm -qi selinux-policy rpm -i selinux-policy-X.XXXXX rpmbuild -bp /rpmbuild/SPECS/selinux-policy.spec
Now that you have downloaded the appropriate policy files, we can start with an example. By default, the apache policy allows for execution of any command. This creates a significant command execution vulnerability and if our web developers are not adept enough to use secure coding practices, we could have a huge breach! So, let's go look at the apache policy files yet again.

We can use some of our policy analysis tools such as apol, sepolicy, and/or seinfo to see if the policy is currently giving access. The problem with this approach is that it is not clear where this access is given. Remember that all the policies are generally compiled and put together in a policy file. The method we will take first, is to simply look through the type enforcement file for any allow rules or interfaces that will allow malicious actions.

```
[root@localhost selinux-policy-contrib-73a88dc7435b803ba860e8938c9611dd62ef6d5c]# pwd
/root/rpmbuild/BUILD/selinux-policy-contrib-73a88dc7435b803ba860e8938c9611dd62ef6d5c
[root@localhost selinux-policy-contrib-73a88dc7435b803ba860e8938c9611dd62ef6d5c]# ls
apache.fc
apache.if
apache.te
[root@localhost selinux-policy-contrib-73a88dc7435b803ba860e8938c9611dd62ef6d5c]#
```

Getting Apache Files from Downloaded Policy

As we move through the policy file, we are greeted with a number of gen_tunables (booleans), and some initial statements defining types, attributes, and calling preliminary interfaces. As we explore these rules and interfaces, you may have a hard time understanding what they do. Remember that interfaces are defined in some other file so it can be helpful to search for them on the system. For example, what does the "files_type" interface do? We can either use our intuition and find the interface file where this comes from (it comes from the files.if interface) or we can just search for all instances of it with grep. Here is a command I use to find where interfaces and other policy lines may be used. We can now see some of the documentation that shows what files_type is used for. Here we see that the interface is used to make a type able to be placed on files. This is a perfect example of an interface abstraction that makes SELinux policy creation easier for us! We shouldn't have to worry about making a new type and then making that type able to be placed on files, it should just work once we create the type label.

```
# httpd_cache_t is the type given to the /var/cache/httpd
# directory and the files under that directory
type httpd_cache_t;
files_type(httpd_cache_t)

# httpd_config_t is the type given to the configuration files
type httpd_config_t;
files_config_file(httpd_config_t)
```

files_type Interface Found in apache.te

```
[root@localhost BUILD]# grep -rnw . -e 'files_type*' | grep 'interface*'
./selinux-policy-contrib-73a88dc7435b803ba860e8938c9611dd62ef6d5c/cups.te:39:files_type(cupsd_interface_t
)
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/contrib/cups.te:39:files_type(cu
psd_interface_t)
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/kernel/files.if:210:interface(`f
iles_type',`
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/kernel/files.if:364:## calls to
files_type() redundant.  Failure to use this interface
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/kernel/files.if:411:## calls to
files_type() redundant.  Failure to use this interface
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/kernel/files.if:539:## calls to
files_type() redundant.  Failure to use this interface
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/kernel/files.if:8558:##       c
alls to files_type() redundant.  Failure to use this interface
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/system/authlogin.if:210:##     o
ther interfaces that permit files_type access.
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/system/logging.if:12:##       c
alls to files_type() redundant.  Failure to use this interface
./selinux-policy-55f4df96a3aff2ed1791e428385e1967856eed49/policy/modules/system/miscfiles.if:11:##     c
alls to files_type() redundant.  Failure to use this interface
[root@localhost BUILD]#
```

Finding where the Interface is Defined

```
########################################
## <summary>
##      Make the specified type usable for files
##      in a filesystem.
## </summary>
## <desc>
##      <p>
##      Make the specified type usable for files
##      in a filesystem.  Types used for files that
##      do not use this interface, or an interface that
##      calls this one, will have unexpected behaviors
##      while the system is running. If the type is used
##      for device nodes (character or block files), then
##      the dev_node() interface is more appropriate.
##      </p>
##      <p>
##      Related interfaces:
##      </p>
##      <ul>
```

file_interface makes a type specific to files

```
##      files_type(myfile_t)
##      allow mydomain_t myfile_t:file read_file_perms;
##      </p>
## </desc>
## <param name="type">
##      <summary>
##      Type to be used for files.
##      </summary>
## </param>
## <infoflow type="none"/>
#
interface(`files_type',`
        gen_require(`
                attribute file_type, non_security_file_type, non_auth_file_type;
        ')

        typeattribute $1 file_type, non_security_file_type, non_auth_file_type;
')
```

file_interface pt. 2

Back to our apache policy hunt. We are looking for some allow rule or interface that may enable the system to execute commands. Since each program generally runs in a different domain from other programs, we can probably assume it's an interface that is causing this issue. The reason for this intuition is that if there wasn't an interface controlling this, there would need to be allow rules for many application domains allowing access. Testing commands, we know we can enter multiple different domains.

Continuously searching through the policy file, I find an interface that seems fishy in what the underlying functionality may be. The interface, shown below, is called application_exec_all. The one parameter supplied is the httpd_t type, which leads me to believe it allows the httpd_t domain to access any command. Let's go find the interface and see exactly what is going on. After searching for and navigating to the interface, we can see that this interface simply gives a domain access to execute every executable on the system. That seems like a pretty big oversight by the authors of the apache policy, especially since there is an interface (application_exec) which seemingly allows you to execute a specific application in the calling domain (has less access than the target domain). So, let's take out this interface out. Additionally, it seems like there are some additional command execution interfaces being used called corecmd_exec_*. As these are also listed in the interface, let's take them out and see what happens when we reload the policy.

```
fs_rw_hugetlbfs_files(httpd_t)
fs_exec_hugetlbfs_files(httpd_t)
fs_list_inotifyfs(httpd_t)

auth_use_nsswitch(httpd_t)

application_exec_all(httpd_t)

# execute perl
corecmd_exec_bin(httpd_t)
corecmd_exec_shell(httpd_t)

domain_use_interactive_fds(httpd_t)
domain_dontaudit_read_all_domains_state(httpd_t)
```

Suspicious Interface found in apache.te

```
########################################
## <summary>
##      Execute all executable files.
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary>
## </param>
## <rolecap/>
#
interface(`application_exec_all',`
        corecmd_dontaudit_exec_all_executables($1)
        corecmd_exec_bin($1)
        corecmd_exec_shell($1)

        application_exec($1)
')
```

Allows execution of any program! - Not great

```
apache.te:536: Warning: mmap_files_pattern() is deprecate
apache.te:1729: Warning: miscfiles_read_certs() has been
rts() instead.
Creating targeted apache.pp policy package
rm tmp/apache.mod.fc tmp/apache.mod
[root@localhost apache_policy_files]# semodule -i apache.
libsemanage.semanage_direct_install_info: Overriding apac
riority 400.
[root@localhost apache_policy_files]#
```

Removed the application_exec_all Interface

```
apache.te:536: Warning: mmap_files_pattern() is deprecate
apache.te:1729: Warning: miscfiles_read_certs() has been
rts() instead.
Creating targeted apache.pp policy package
rm tmp/apache.mod.fc tmp/apache.mod
[root@localhost apache_policy_files]# semodule -i apache.
libsemanage.semanage_direct_install_info: Overriding apac
riority 400.
[root@localhost apache_policy_files]#
```

Compiled the new apache policy

It looks as if we have stopped the command execution but once broke our pinging that we fixed earlier! Just reload our old ping policy (or create a new one), and load it at a higher priority. For reference policies loaded by users are loaded at 400. So, we can just give our ping policy a 500 priority like so.

```
[root@localhost ping]# semodule -X 500 -i apache_ping.pp
[root@localhost ping]# semodule --list-modules=full | hea
500 apache_ping       pp
400 apache            pp
400 demoadmin         pp
400 demoadmin_login   pp
400 testadmin         pp
400 testadmin_login   pp
200 flatpak           pp
100 abrt              pp
100 accountsd         pp
100 acct              pp
[root@localhost ping]#
```

Removed the application_exec_all Interface

Please note that if you have any additional issues, continue the troubleshooting steps we used before to build a module that will allow the actions you are trying to take. I had to perform these steps and ended with a policy module looking like the following.

```
module new_mod 1.0;

require {
        type ping_exec_t;
        type httpd_t;
        class file { execute open read };
        class file execute_no_trans;
        class file map;
        type httpd_sys_script_t;
        class rawip_socket { read write };
        class process setcap;
}

allow httpd_t ping_exec_t:file open;
allow httpd_t ping_exec_t:file { execute read };
allow httpd_t ping_exec_t:file execute_no_trans;
allow httpd_t ping_exec_t:file map;
allow httpd_sys_script_t self:process setcap;
allow httpd_t self:rawip_socket { read write };
~
```

Created a new module via troubleshooting

5.4.2 Using a Constrain Policy Module

This section will include a constrain rule that seemingly blocks most if not all bin_t file reads, which are needed to execute standard commands on Linux. It is my belief that the CIL standard of writing policies needs more resources devoted towards proper usage. There are a number of resources online that will be linked in the Further Reading Section. Essentially, the goal we are trying to achieve is to prevent the apache process from reading any bin_t binaries. Using constrains to prevent access can be a quick and powerful tool for locking down specific system interactions that could be attributed to an attack.
So, I have written a constrain statement in the CIL language that seemingly accomplishes this goal. We want to constrain on file objects with an read access requests. Furthermore, anytime the source type is httpd_t and the target type is not bin_t we want to allow file reads. In addition, when the source type is not httpd_t we want to allow as well, this ensures that we don't accidentally stop all other access requests on the system. It can be a bit complex to wrap your head around, but it is essentially saying that if we are dealing with httpd_t to bin_t transitions we want to deny access.

```
(constrain (file (read))
(or
        (and
            (eq t1 httpd_t)
            (not(eq t2 bin_t))
        )
        (not(eq t1 httpd_t))
)
)
~
```

Created a new constrain rule

5.5 Creating a New Policy for a Custom Application

In this section, we will discuss what creating a policy based on some application running as a service looks like. SELinux aims to make this process as smooth as possible. First, we have a simple daemon that returns the text "Message", to anyone that connects along the 5004 TCP port.

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
        int listenfd = 0, connfd = 0;
        struct sockaddr_in server_address;


        char sendBuffer[1025];
        time_t ticks;

        listenfd = socket(AF_INET,SOCK_STREAM,0);
        memset(&server_address, '0', sizeof(server_address));
        memset(sendBuffer, '0', sizeof(sendBuffer));

        server_address.sin_family = AF_INET;
        server_address.sin_addr.s_addr = htonl(INADDR_ANY);
        server_address.sin_port = htons(5004);

        bind(listenfd, (struct sockaddr*)&server_address, sizeof(server_address));

        listen(listenfd, 10);

        while(1)
        {
```

server_code.c pt 1

```
        server_address.sin_family = AF_INET;
        server_address.sin_addr.s_addr = htonl(INADDR_ANY);
        server_address.sin_port = htons(5004);

        bind(listenfd, (struct sockaddr*)&server_address, sizeof(server_address));

        listen(listenfd, 10);

        while(1)
        {
            connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

            strcpy(sendBuffer, "Message");
            write(connfd, sendBuffer, strlen(sendBuffer));

            close(connfd);
            sleep(1);
        }
    }
```

server_code.c pt 2

Once you have your code, make sure to compile it. Here I am just going to compile it and rename the output file. We will also turn this into a daemon but creating a service file. In this service file we are just telling the kernel to run this program in the background and where it lives.

```
[Unit]
Description=Server for testing

[Service]
Type=server
ExecStart=/usr/local/bin/simple_serv

[Install]
WantedBy=multi-user.target
```

Created a service file

Compile the binary, copy files, and start the service

```
$ gcc –o simple_serv server_code.c
$ cp simple\_serv /usr/local/bin
$ cp simple\_serv.service /usr/lib/systemd/system
$ systemctl start simple\_serv
$ systemctl status simple\_serv$
```

As we can see, the daemon runs unconfined and we want to change this. So, we will use the sepolicy command to generate a policy based on our daemon.

```
[root@localhost test_app]# ps -efZ | grep simple
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 cullen 2983 2769  0 22:59 tty2 00:00:08 /usr/libexe
c/ibus-engine-simple
system_u:system_r:unconfined_service_t:s0 root 36357   1  0 23:44 ?        00:00:00 /usr/local/bin/simple
_serv
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 root 36367 3331  0 23:44 pts/0 00:00:00 grep --colo
r=auto simple
```

Daemon running unconfined

```
[root@localhost test_app]# sepolicy generate --init /usr/local/bin/simple_serv
Created the following files:
/home/cullen/test_app/simple_serv.te # Type Enforcement file
/home/cullen/test_app/simple_serv.if # Interface file
/home/cullen/test_app/simple_serv.fc # File Contexts file
/home/cullen/test_app/simple_serv_selinux.spec # Spec file
/home/cullen/test_app/simple_serv.sh # Setup Script
```

Creating policy

We can use the helper script to compile and load the policy automatically, and we can now see that the program is being confined properly. From here, we can look out for any alerts and debug them as well as add our own rules to the policy files and reload the module!

```
[root@localhost test_app]# systemctl restart simple_serv.service
[root@localhost test_app]# ps -efZ | grep simple
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 cullen 2983 2769  0 22:59 tty2 00:00:08 /usr/libexe
c/ibus-engine-simple
system_u:system_r:simple_serv_t:s0 root    36585       1  0 23:47 ?        00:00:00 /usr/local/bin/simple
_serv
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 root 36605 3331  0 23:48 pts/0 00:00:00 grep --colo
r=auto simple
```

Daemon running confined

6

SECTION

Bibliography

The following environment and command creates a clean and organized bibliography.

```
\begin{thebibliography}{numberofsources}
  ⟨text⟩
\end{thebibliography}
```

The following environment is needed to create the bibliography.

`\bibitem`

The following command creates each source.

6.1 Bibliography Source Code

Spacing is just used to help show the different commands and environments. Do not feel obligated to include the spacing exactly how it is.

```
\begin{thebibliography}{5 % - Number of Sources You are Citing}

\bibitem{financialbrand}Jim, M. (2018). \emph{10 Technologies That
Will Disrupt Financial Services In The Next 5 Years.}\\
Retrieved from:\\
\url{https://thefinancialbrand.com/77228/technology-trends-disrupting
\\-financial-services-banking-future/}

\bibitem{Forbes} Alan, M. (2018). \emph{How Banks Should Navigate
as Their North Star Shifts.}\\
Retrieved from:\\
\url{https://www.forbes.com/sites/alanmcintyre/2018/10/18/how-banks-should-
navigate-through-digital-disruption-as-their-north-star\\-shifts/#39c502223ee7}

\bibitem{americanbanker} Penny, C., Will, H., Suleman, D. (2019). \emph{10 way
technology will change banking in 2019.}\\
Retrieved from:\\
\url{https://www.americanbanker.com/list/10-ways-technology-will-change
-banking\\-in-2019}

\bibitem{Robert Wright} Wright, Robert. (2008). \emph{Origins of Commercial Banking
in the United States, 1781-1830. EH.Net Encyclopedia.}\\
Retrieved from:\\
\url{http://eh.net/encyclopedia/origins-of-commercial-banking-in-the
\\-united-states-1781-1830/}

\bibitem{The Economist} \emph{Tech's raid on the banks.} (2019).
The Economist, 431(9141), 9.

\end{thebibliography}
```

7

SECTION

Further Reading

7.1 Websites and Tutorials

If you are interested in understanding how SELinux really works, I highly encourage reading the following package pdfs/websites/tutorials:
If you are looking for a solution for a problem you have encountered and can not find it within these files, see the templates created in the folder that orignially contained this pdf file or contact me at: cxr5971@gmail.com